Efficient Parallel and Fast Convergence Chaotic Jaya Algorithms

H. Migallón[a,*], A. Jimeno-Morenilla[b], J.L. Sánchez-Romero[b], A. Belazi[c]

[a]*Department of Computer Engineering, University Miguel Hernández, E-03202, Elche, Alicante, Spain.*
[b]*Department of Computer Technology, University of Alicante, E-03071, Alicante, Spain.*
[c]*RISC Laboratory National Engineering School of Tunis, University of Tunis El Manar, Tunis, Tunisia*

Abstract

The Jaya algorithm is a recent heuristic algorithm for solving optimisation problems. It involves a random search for the global optimum, based on the generation of new individuals using both the best and the worst individuals in the population, thus moving solutions towards the optimum while avoiding the worst current solution. In addition to its performance in terms of optimisation, a lack of control parameters is another very important advantage of this algorithm. However, the number of iterations needed to reach the optimal solution, or close to it, may be very high, and the computational cost can hamper compliance with time requirements. In this work, a chaotic 2D map is used to accelerate convergence, and parallel algorithms are developed to alleviate the computational cost. The parallel algorithms developed here are based both on a multipopulation structure and on an improved (computational) use of the chaos map.

Keywords: optimisation, Jaya algorithm, chaotic map, parallel algorithms, OpenMP

---

1. Introduction

Optimisation algorithms are used to find the optimal value, or a value as close to this as possible, for a given function called the cost function. Depending on the intrinsic characteristics of the cost function, finding this value can be a challenge, and depending on the search pattern used, the optimisation algorithm can be trapped in local

*Corresponding author
Email address: hmigallon@umh.com (H. Migallón)

optimums. Population-based algorithms, such as the one considered in this work, are also iterative algorithms, and depending on the number of iterations to be performed, the computational cost can increase dramatically.

When deterministic methods are applied to solve an optimisation problem, a sequence of points tending to the optimal value is generated based on the analytical properties of the problem to be solved. In this case, the optimisation problem becomes a problem of linear algebra, i.e. the gradient of the cost function is used in many cases to solve the optimisation problem. Although deterministic methods can be used to solve the problems of optimisation of many functions [1], for large-scale problems, and especially non-differentiable, non-convex and nonlinear objective functions, deterministic methods are either unable to reach the solution or the computational cost prevents their use. A number of heuristic methods have been proposed to overcome these drawbacks, where the solution obtained is acceptable and the computational cost is reasonable. In most cases, meta-heuristic methods employ guided search techniques in which certain random processes are used to solve the problem. Although it cannot be formally demonstrated that the optimum value obtained is the solution to the problem, they have been shown via experiment to be robust.

In the past few decades, several well-known meta-heuristic optimisation algorithms based on natural phenomena have been proposed: for example, the particle swarm optimisation (PSO) algorithm [2] and its variants are based on the social behaviour of fish schooling or bird flocking; the artificial bee colony (ABC) algorithm [3] was inspired by the foraging behaviour of honey bees; the shuffled frog leaping (SFL) [4] algorithm imitates the collaborative behaviour of frogs; the ant colony optimisation (ACO) algorithm [5] imitates the foraging behaviour of ant colonies; the evolutionary strategy (ES) algorithm [6] is based on the processes of mutation and selection seen in evolution; genetic programming (GP) [7] and evolutionary programming (EP) [8] are techniques for evolving programs based on the selection of individuals for reproduction (crossover) and mutation; the firefly (FF) algorithm [9] was inspired by the flashing behaviour of fireflies; the gravitational search algorithm (GSA) [10] was based on Newtons law of gravity; the biogeography-based optimisation (BBO) algorithm [11] improves solutions stochastically and iteratively; the grenade explosion method (GEM) algorithm

[12] is based on the characteristics of the explosion of a grenade; genetic algorithms (GA) [13] and their variants reflect the process of natural selection; the artificial immune algorithm (AIA) [14] is based on the behaviour of the human immune system; differential evolution (DE) [15] and its variants attempt to iteratively improve a candidate solution with respect to a given measure of quality; the simulated annealing (SA) algorithm [16] is based on the annealing process in metallurgy; the tabu search (TS) algorithm [17] employs meta-heuristic local search methods; the teaching-learning-based optimisation (TLBO) algorithm [18] is based on the processes of teaching and learning; and the harmony search algorithm (HSA) [19] was inspired by the process of musical performance.

None of these algorithms are free of limitations in terms of their evolution process, and some of them are easily trapped in local minima. However, a key aspect of the behaviour of these algorithms is the correct adjustment of the control parameters, since the effectiveness of these algorithms depends heavily on the correct setting of these fixed control parameters [18]; for example, PSO needs the cognitive and social parameters and inertia weights to be adjusted; GA needs the crossover probability, mutation probability, selection operator, etc. to be set; the SA algorithm needs the initial annealing temperature and cooling schedule to be tuned; BBO needs the probability of habitat modification, mutation probability, habitat elitism parameter and population size to be set; ABC needs the number of bees and limits to be defined; HSA needs the harmony memory consideration rate, the number of improvisations, etc. to be adjusted; and BBO needs the immigration rate, emigration rate, etc. to be set. In contrast, both TLBO and the Jaya optimisation algorithm used in this work can overcome this drawback, since both algorithms only need general parameters to be established, such as the population size and number of iterations (or number of generations) or the stopping criteria.

These algorithms are suitable for solving large-scale industrial problems. For example, in [20] authors take advantage of the lack of control parameters of the Jaya algorithm to optimise the coefficients of proportional plus integral controller and filter parameters of photovoltaic fed distributed static compensator, where Jaya improves the performance of the TLBO, note that the latter also lacks control parameters; and in [21]

Jaya is modified for efficiently solving the maximum power point tracking problem of photovoltaic systems. In [22], the ABC algorithm is used to solve the welded beam problem, the pressure vessel problem, the tension-compression spring problem, the speed reducer design and the gear train design. Jaya has also been used, for example, to optimise the automatic application of carbon fiber reinforced polymer composites in various production processes [23]; to analyse the effects of the parameters of the submerged arc welding process on the geometry of the weld seam [24]; to optimise sensor placement and damage identification in laminated composite structures [25]; to design a proportional-integral-derivative controller for automatic generation control of an interconnected power system [26]; to identify the parameters of photovoltaic models used for simulation, evaluation and control of such systems; etc. It can also be used to solve problems commonly used in industrial applications, for example for generalized sparse non-negative matrix factorization [27] or for matrix factorization methods, applied in [28, 29] to large-scale collaborative filtering recommender systems. Moreover, parallel optimisation algorithms have been used, for example, to optimise the tool path of numerical control machines [30]; to allocate generation and transmission resources in an electricity market [31]; or to control the flow distribution generated by the heliostat field of the receiving system of a solar power plant [32].

Two of the most widely used techniques for improving these algorithms are hybridisation and the use of chaos theory. Hybridisation involves the use of more than one search technique to improve the behaviour of the optimisation process (see for example [33, 34, 35, 36, 37]). This hybridisation, as it is logical, usually improves the quality of the solution obtained. However, it may make it difficult to adjust the parameters correctly and can also increase the computational cost. Chaos theory concerns the study of chaotic dynamical systems, which can be defined as nonlinear dynamical systems that are characterised by a high sensitivity to their initial conditions [38, 39]. Many of the algorithms mentioned here make use of randomness in their search patterns, and this randomness can be totally or partially replaced by the use of chaos theory, which offers a powerful technique for hybridisation. Chaos has been used in meta-heuristic algorithms to: (a) replace random number sequences with sequences generated by chaotic maps; (b) perform a local search by means of a chaotic map function; and (c) to gener-

ate the control parameters in a chaotic way [40].

Some of the works in which chaos has been successfully used to improve metaheuristic optimisation algorithms include the following: in [41, 42], chaos was applied to the FF algorithm; in [43, 44] it was applied to GA algorithms; in [45, 46], it was applied to the SA algorithm; in [47, 48], it was applied to the BBO algorithm; in [49, 50], it was applied to the DE algorithm; in [51], it was applied to ABC algorithm; and in [52], it was applied to the GSA algorithm.

The main objective of our work is to develop efficient parallel algorithms based on the chaotic C-Jaya algorithm. The aim is for these algorithms to be efficient both at the level of optimisation behaviour and at the level of computational cost. The parallel algorithms developed here are based on both multi-population-based algorithms, a technique already used in several sequential and parallel proposals (see for example [53, 54, 55, 56]), and on the 2D chaotic map presented in [57]. Parallel algorithms are computationally improved and adapted to the use of the 2D chaotic map.

The major contributions of this paper are summarised below. Firstly, we analyse the use of the 2D cross chaotic map proposed in the Jaya algorithm, both at computational level and at optimisation behaviour level. The results show a significant improvement in optimisation performance, but at the expense of a huge increase in the computational complexity during the optimisation process. Secondly, to reduce the global computing time, we develop efficient parallel algorithms based on multi-populations, in which the increase in computational complexity introduced by the use of the 2D chaotic map reduces the intrinsic parallelism that can be exploited, thus reducing parallel scalability. Finally, we modify the pattern of use of the chaotic map to develop more efficient and scalable parallel algorithms, which maintain the remarkable improvement in optimisation behaviour.

The remainder of this paper is organised as follows: Section 2 presents a brief description of the Jaya algorithm and the 2D chaotic map used here. In Section 3, the parallel algorithms and the improvements we introduce are explained in detail. In Section 4 we analyse the performance of the proposed parallel algorithms, and finally, in Section 5, some conclusions are drawn.

2. Preliminaries

130 The work presented in the article is based both on the Jaya algorithm presented in Section 2.1 and on the use of a 2D chaotic map presented in Section 2.2.

2.1. Jaya algorithm

The Jaya algorithm was presented in [58], in which the results of optimising both constrained and unconstrained functions are reported. These results show that the Jaya 135 algorithm behaves better than the most common reference algorithms, and a more detailed comparative analysis is presented in [59]. The Jaya algorithm is a population-based algorithm in which the evolution does not depend on function-specific adjustment parameters, and only the size of the population and the maximum number of evaluations to be performed need to be set.

140 Once the best and worst individuals of the current population have been identified, the basic operation of the Jaya algorithm consists of a search for the global optimum, which is achieved by moving toward the best individual and avoiding the worst. This strategy is implemented by obtaining a new individual following Equation (1), in which $r_{1,j}$ and $r_{2,j}$ are uniformly distributed random numbers, and j refers to the design 145 variable of the specific cost function to be optimised.

$$x_j^{'} = x_j + r_{1,j} \left(x_{j,best} - |x_j| \right) - r_{2,j} \left(x_{j,worst} - |x_j| \right) \quad (1)$$

Jaya is an iterative algorithm in which Equation (1) is applied at each iteration to each individual in the current population. If the new individual is better than the old one, it is exchanged; otherwise, it is discarded. Algorithm 1 describes this pro-150 cess. Different executions are performed in this type of algorithm, since the intrinsic randomness of these algorithms may cause unsatisfactory execution (see line 2). A random initial population is computed (lines 3–8). At each iteration (line 9), after searching for the best and worst individuals in the population (line 10), new individuals are computed, and are inserted into the population as replacements if they are better

155 than the previous ones. The value of each variable for the new individual is trimmed if necessary (lines 16–21).

The use of chaotic maps is proposed in order to increase the diversity of the population, and thus avoid a possible premature convergence at a local minimum and accelerate the convergence. When a chaotic map is used, the randomness of Equation (1)
160 (provided by the two random numbers) is replaced by a sequence of numbers that can be generated from a random starting point. In this work, we use the chaotic map presented in [57], and this is briefly described in Section 2.2.

2.2. 2D chaotic map

The 2D chaotic map used here [57] balances the exploitation and exploration phases
165 that are characteristic of heuristic optimisation algorithms. Note that the exploitation phase is related to the convergence ratio, while the exploration phase is related to the algorithms ability to explore different regions in a search space. In order to balance both of these phases, the new individuals in Algorithm 1 are obtained by using a random individual (x^{rand}) from the current population, in addition to the current best and
170 current worst individuals. Moreover, the new individual can be obtained in three different ways, using Equations (2), (3) and (4). In these equations, as described above, x^{rand} is a random individual, and $x_{_,best}$ and $x_{_,worst}$ are the current best and worst individuals, respectively, of the population. Each $ch_{_,j}$ is the absolute value of a chaotic variable from the 2D cross chaotic map, and S_F is a scaling factor that takes a value of
175 one or two.

$$x_j^{'} = ch_{1,j} x_j^{rand} + ch_{2,j} \left(x_j - ch_{3,j} x_j^{rand} \right) + ch_{4,j} \left(x_{j,best} - ch_{5,j} x_j^{rand} \right) \quad (2)$$

$$x_j^{'} = ch_{1,j} x_j^{rand} + ch_{2,j} \left(x_j - ch_{3,j} x_j^{rand} \right) + ch_{4,j} \left(x_{j,worst} - ch_{5,j} x_j^{rand} \right) \quad (3)$$

$$x_j^{'} = ch_{1,j} x_{j,best} + ch_{2,j} \left(x_j^{rand} - S_F x_{j,best} \right) \quad (4)$$

Algorithm 1 Jaya algorithm

1: Set parameters ($Iterations$ and $PopulationSize$) and define cost function
2: **for** $l = 1$ to $Runs$ **do**
3:   **for** $i = 1$ to $PopulationSize$ {Create Initial Population X:} **do**
4:     **for** $j = 1$ to $VARS$ **do**
5:       $x_j^i = MinValue + (MaxValue - MinValue) * rand_{[0,1]}$
6:     **end for**
7:     Compute and store $F(x_j^i)$ {Function evaluation}
8:   **end for**
9:   **for** $l = 1$ to $Iterations$ **do**
10:     Search for $best$ and $worst$ individuals
11:     **for** $i = 1$ to $PopulationSize$ {Create New Population X':} **do**
12:       **for** $j = 1$ to $VARS$ **do**
13:         Obtain 2 random numbers ($rand_{1,2_{[0,1]}}$)
14:         $x_j^{'i} = x_j^i + rand_{1,j}\left(x_{j,best}^i - \left|x_j^i\right|\right) - rand_{2,j}\left(x_{j,worst}^i - \left|x_j^i\right|\right)$
15:         { Check the bounds of $x_j^{'i}$ }
16:         **if** $x_j^{'i} < MinValue$ **then**
17:           $x_j^{'i} = MinValue$
18:         **end if**
19:         **if** $x_j^{'i} > MaxValue$ **then**
20:           $x_j^{'i} = MaxValue$
21:         **end if**
22:       **end for**
23:       Compute $F(x^{'i})$ {Function evaluation}
24:       **if** $F(x^{'i}) < F(x^i)$ **then**
25:         Replace solution in population
26:       **end if**
27:     **end for**
28:   **end for**
29: **end for**
30: Obtain Best Solution and Statistical Data

The construction of the 2D chaotic map is shown in Algorithm 2. In the experiments performed here, the initial conditions for generating the 2D chaotic map are $x_1 = 0.2$, $y_1 = 0.3$, $k = i$ and $maxDimMap = 500$. The computed values x_i and y_i are in the range $[-1, 1]$.

Algorithm 2 2D Chaotic map

1: Initialize x_1
2: Initialize y_1
3: Initialize $maxDimMap$
4: **for** $i = 1$ to $maxDimMap$ **do**
5: $\quad x_{i+1} = cos(k * arccos(y_i))$
6: $\quad y_{i+1} = 16x_i^5 - 20x_i^3 + 5x_i$
7: **end for**

180 In each iteration of Algorithm 1, the three possible options (Equations (2), (3) and (4)) to obtain the new iterate are not all computed, and only one of them is used at each iteration, as shown in Algorithm 3.

It is also worth mentioning that the initial population is not computed as in the original Jaya algorithm (see lines 4–6 of Algorithm 1), and instead uses the chaotic 185 map, as shown in lines 3–9 of Algorithm 4.

3. Proposed parallel algorithms

The parallel algorithms developed here are based on the use of sub-populations. However, due to the computational characteristics of the chaotic algorithm, it is not possible to obtain high efficiency using the same parallel strategies as those used in 190 [54]. It can be predicted, as discussed in Section 2.2, that the computational cost per iteration will depend on the computational cost of the function to be optimised, the selection of the chaotic values to be used and their extraction from the chaotic map.

A general flowchart of the parallel algorithms developed here is shown in Figure 1. The most important steps and improvements introduced to accelerate the algorithm are 195 detailed below.

Algorithm 3 Selection of the population update option

1: Obtain two ordered integer random numbers and one chaotic number:
2: $a = min$(rnd1,rnd2)
3: $b = max$(rnd1,rnd2)
4: ch_j are randomly selected chaotic values
5: Select between equations (2), (3) or (4):
6: **if** $ch_j < a$ **then**
7: $x_j^{'} = ch_{1,j}x_j^{rand} + ch_{2,j}\left(x_j - ch_{3,j}x_j^{rand}\right) + ch_{4,j}\left(x_{j,best} - ch_{5,j}x_j^{rand}\right)$
8: **end if**
9: **if** $a < ch_j < b$ **then**
10: $x_j^{'} = ch_{1,j}x_j^{rand} + ch_{2,j}\left(x_j - ch_{3,j}x_j^{rand}\right) + ch_{4,j}\left(x_{j,worst} - ch_{5,j}x_j^{rand}\right)$
11: **end if**
12: **if** $ch_j > b$ **then**
13: $x_j^{'} = ch_{1,j}x_{j,best} + ch_{2,j}\left(x_j^{rand} - S_F x_{j,best}\right)$
14: **end if**

First, we analyse the decisions made regarding the efficient use of memory. Our algorithms are based on sub-populations, and all of them divide the individuals of the whole population among the available computing processes, thus creating sub-populations. It is worth noting that these algorithms are executed on a multicore com-
200 puting platform, i.e., in a shared memory computer architecture. However, each process (or thread) in the initial step copies the sub-population that has been assigned to it to its local private memory. In contrast, the chaotic optimisation algorithm considered here is based on a 2D cross chaotic map, for which the calculation is shown in Algorithm 2. To ensure efficient behaviour of the parallel algorithms, the chaotic map must
205 be stored in global memory, and there is no overhead due to contention in memory writing.

In Algorithm 3, up to three individuals (the best, worst and a random individual) from the current population may be needed to generate a new individual. Since we avoid using extra memory to store the new sub-population, a copy of these individ-
210 uals must be stored during the generation of the new sub-population, allowing these

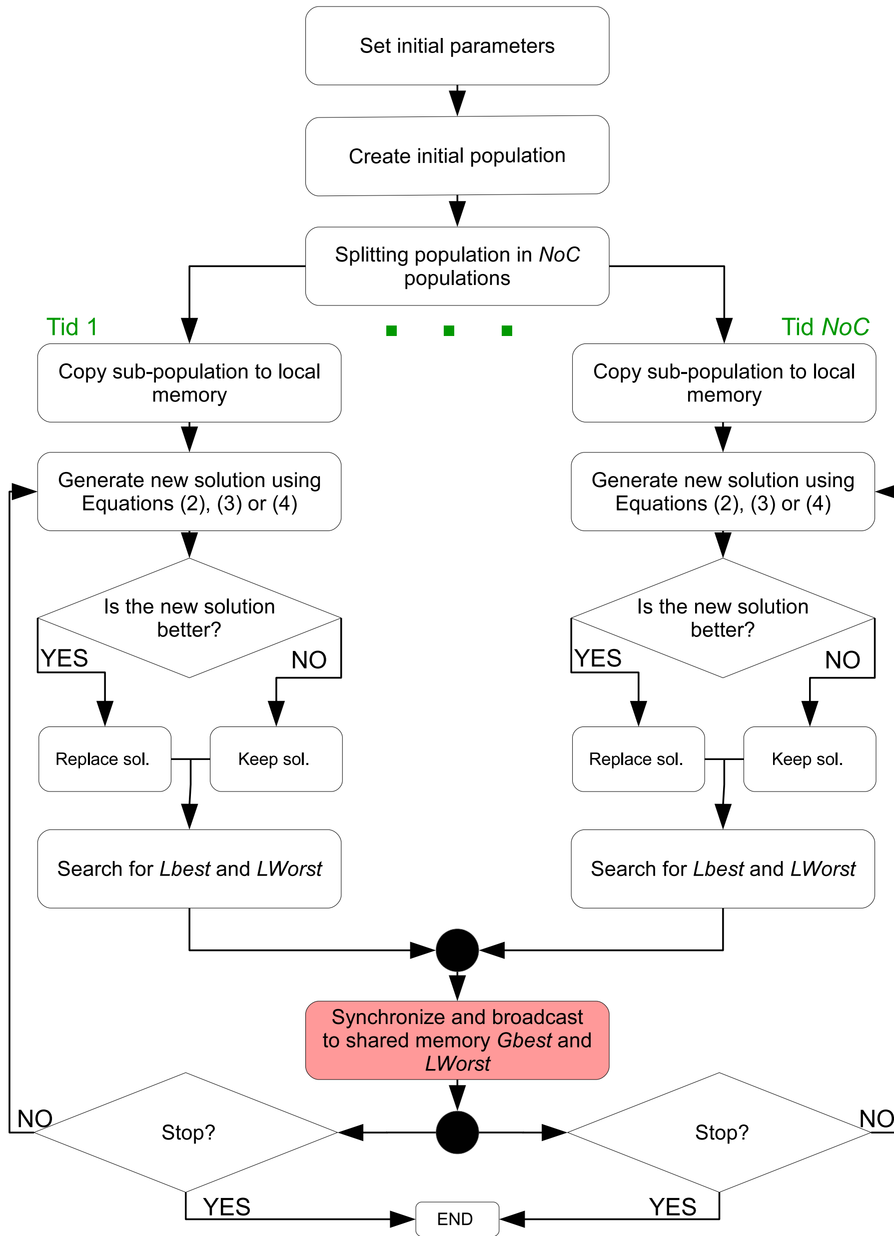

Figure 1: General flowchart of the proposed parallel algorithms.

new individuals to replace others in the same sub-population if they represent improvements. Algorithm 4 shows the computation of both the initial population and the sub-

population sizes. Both processes are performed sequentially before the parallel region is spawned.

Algorithm 4 Computing of the initial population and sub-population sizes

1: Set parameters ($Iterations$ and $PopulationSize$) and define cost function
2: Set the number of computing processes (NoC)
3: **for** $i = 1$ to $PopulationSize$ **do**
4:     **for** $j = 1$ to $VARS$ **do**
5:         Obtain rnd: random integer value in range $[1, maxDimMap]$
6:         $x_j^i = MinValue + (MaxValue - MinValue) * ch(rnd)$
7:     **end for**
8:     Compute and store $F(x^i)$
9: **end for**
10: $SubPopSize = PopulationSize/NoC$
11: **for** $i = 1$ to NoC **do**
12:     $SubPopSizeArray[i] = SubPopSize$
13:     **if** $i <= (PopulationSize\%NoC)$ **then**
14:         $SubPopSizeArray[i] + +$
15:     **end if**
16: **end for**

The first step in the parallel region is to copy the assigned sub-population to local memory, and then to search for the best and worst individuals in the sub-population. Algorithm 5 shows how these processes are performed within a given parallel region.

3.1. CP-CJaya parallel algorithm

Two different parallel strategies for accelerating the optimisation algorithm were developed. In the first one, called the CP-CJaya (Communicated Parallel Chaotic Jaya) algorithm, the different processes share information, and coordination processes between them are therefore necessary. Hence, in this strategy, as can be seen in Algorithm 6, after having searched for the best and worst individuals in each sub-population, all the threads must be coordinated in order to select the global best and global worst in-

Algorithm 5 Copy of sub-population and search for the best and the worst.

1: Inside a parallel region:
2: Identify thread Tid in range $[1, NoC]$
3: $MySubPopSize = SubPopSizeArray[Tid]$
4: Allocate memory in private memory for population of size $MySubPopSize$
5: $IniSubPop = 0$
6: **for** $i = 1$ to $Tid - 1$ **do**
7: $\quad IniSubPop+ = SubPopSizeArray[i]$
8: **end for**
9: $EndSubPop = IniSubPop + SubPopSize$
10: Copy sub-population ($IniSubPop - EndSubPop$) into private memory
11: Search for local best ($LBest$) and local worst ($LWorst$)

225 dividuals. Copies of both individuals are stored in global memory to allow access by all threads. Algorithm 6 includes two synchronisation points (lines 11 and 24), and between these two points there are two critical regions in which the code is executed sequentially by all threads, i.e. with mutual exclusion. As mentioned above, up to three individuals can be used to create a new generation, i.e. in addition to the best and worst, 230 another individual is randomly chosen. In the latter case, in order to avoid increasing the number of communications and coordination processes, it is randomly chosen by each thread from the individuals in its sub-population, and is stored in private memory.

The communications and coordination processes in Algorithm 6 cause a loss of parallel efficiency if they are performed at each iteration, and to solve this problem, 235 we include flags to detect whether it is necessary to update the best or worst global individual. As shown in Algorithm 7, if the best global individual is to be updated, it is not necessary to include synchronisation, and only a critical region is needed; however, a synchronisation point is needed in order to check whether the worst global individual needs to be updated, and this point cannot be removed (line 20 of Algorithm 8). If the 240 worst global individual is to be updated, the process includes a search for the worst local individual, two synchronisation points (lines 24 and 30) and a critical region; these procedures are only performed if the flag F_G_GWorst is set.

Algorithm 6 CP-CJaya: initial search for the global best and global worst.

1: Allocate memory in shared memory for $GBest$ (Global Best)
2: Allocate memory in shared memory for $GWorst$ (Global Worst)
3: Inside a parallel region:
4: Identify thread Tid in range $[1, NoC]$
5: Allocate memory in private memory for x^{rand}
6: Obtain rnd: random integer value in range $[1, MySubPopSize]$
7: Copy rnd individual to x^{rand}
8: Master thread:
9: Copy $LBest_{Tid}$ to $GBest$
10: Copy $LWorst_{Tid}$ to $GWorst$
11: Sync Barrier
12: All threads in parallel CRITICAL region:
13: {
14: **if** $LBest_{Tid}$ is better than $GBest$ **then**
15: Copy $LBest_{Tid}$ to $GBest$
16: **end if**
17: }
18: All threads in parallel CRITICAL region:
19: {
20: **if** $LWorst_{Tid}$ is worse than $GWorst$ **then**
21: Copy $LWorst_{Tid}$ to $GWorst$
22: **end if**
23: }
24: Sync Barrier

Algorithm 7 CP-CJaya: search for the global best based on flags.

1: Flag to find the worst in shared memory: $F_G_GWorst = false$
2: Inside a parallel region:
3: Identify thread Tid in range $[1, NoC]$
4: Flag to find the best in private memory: $F_P_Gbest = false$
5: **for** $i = 1$ to $SubPopSize$ **do**
6: Generate $x^{'i}$
7: **if** $F(x^{'i}) < F(x^i)$ **then**
8: Replace solution in population
9: **if** $F(x^{'i}) < F(LBest_{Tid})$ **then**
10: Update $LBest_{Tid}$
11: **if** $F(x^{'i}) < F(GBest)$ **then**
12: $F_P_Gbest = true$
13: **end if**
14: **end if**
15: **if** $i == LWorst_{Tid}$ **then**
16: $F_G_GWorst = true$
17: **end if**
18: **end if**
19: **end for**
20: **if** $F_P_Gbest == true$ **then**
21: CRITICAL region:
22: **if** $F(LBest_{Tid}) < F(GBest)$ **then**
23: Copy $LBest_{Tid}$ to $GBest$
24: **end if**
25: $F_P_GBest = false$
26: **end if**

Algorithm 8 CP-CJaya: search for the global worst based on flags.

1: Flag to find the worst in shared memory: $F_G_GWorst = false$
2: Inside a parallel region:
3: Identify thread Tid in range $[1, NoC]$
4: Flag to find the best in private memory: $F_P_Gbest = false$
5: **for** $i = 1$ to $SubPopSize$ **do**
6: Generate $x^{'i}$
7: **if** $F(x^{'i}) < F(x^i)$ **then**
8: Replace solution in population
9: **if** $F(x^{'i}) < F(LBest_{Tid})$ **then**
10: Update $LBest_{Tid}$
11: **if** $F(x^{'i}) < F(GBest)$ **then**
12: $F_P_Gbest = true$
13: **end if**
14: **end if**
15: **if** $i == LWorst_{Tid}$ **then**
16: $F_G_GWorst = true$
17: **end if**
18: **end if**
19: **end for**
20: Sync Barrier
21: **if** $F_G_GWorst == true$ **then**
22: Search for $LWorst_{Tid}$
23: SINGLE thread: Copy $LWorst_{Tid}$ to $GWorst$
24: Sync Barrier
25: CRITICAL region:
26: **if** $F(LWorst_{Tid}) > F(GWorst)$ **then**
27: Copy $LWorst_{Tid}$ to $GWorst$
28: **end if**
29: SINGLE thread: $F_G_GWorst = false$
30: Sync Barrier
31: **end if**

3.2. NCP-CJaya parallel algorithm

The second strategy, called the NCP-CJaya (Non-Communicated Parallel Chaotic Jaya) algorithm, was developed with the goal of removing all synchronisation points. Algorithm 9 shows the NCP-CJaya algorithm, in which the three individuals used to create a new generation are stored in private memory. Hence, no global memory space is used except that used to store the chaotic map, as mentioned above.

In both strategies, CP-CJaya and NCP-CJaya, domain decomposition is implemented in order to develop parallel algorithms. Load balancing is especially important in the CP-CJaya algorithm, as the synchronisation barriers can lead to high idle times if the load is not properly balanced. In contrast, NCP-CJaya, which does not include synchronisation barriers, is more versatile and allows load balancing techniques to be applied.

3.3. Improved computing performance (ICP) technique

The use of the chaotic map presented in Section 2.2 increases the computational cost of the Chaotic Jaya sequential algorithm compared to the original Jaya sequential algorithm. The selection of the population update option shown in Algorithm 3, leads to computational cost increase and may decrease the parallel performance of the parallel algorithms proposed.

It should be noted that the random numbers a and b used in Algorithm 3 are calculated before each new individual is generated, while the other five random values are obtained to compute of each variable for each new individual. In order to reduce the computational cost and improve the parallel efficiency in computing each variable for each new individual, only one new chaotic number is extracted from the chaotic map, and the other four are reused. In this way, only one new random value needs to be obtained for the extraction of only one chaotic value. This modification in the use of the chaotic map is called ICP (Improve Computational Performance), and it is shown in Algorithm 10.

Algorithm 9 NCP-CJaya: parallel region without synchronizations.

1: Inside a parallel region:
2: Allocate memory in private memory for $LBest$ (Local Best)
3: Allocate memory in private memory for $LWorst$ (Local Worst)
4: Allocate memory in private memory for x^{rand}
5: Flag to find the worst in private memory: $F_P_LWorst = false$
6: Flag to find the best in private memory: $F_P_LBest = false$
7: Find and store $LBest$ and $LWorst$
8: Obtain rnd: random integer value in range $[1, MySubPopSize]$
9: Copy individual to x^{rand}
10: **for** $i = 1$ to $SubPopSize$ **do**
11:     Generate $x^{'i}$
12:     **if** $F(x^{'i}) < F(x^i)$ **then**
13:         Replace solution in population
14:         **if** $F(x^{'i}) < F(LBest_{Tid})$ **then**
15:             Update $LBest_{Tid}$
16:             $F_P_LBest = true$
17:         **end if**
18:         **if** $i == LWorst_{Tid}$ **then**
19:             $F_P_LWorst = true$
20:         **end if**
21:     **end if**
22:     **if** $F_P_LBest == true$ **then**
23:         Update $LBest$
24:         $F_P_LBest = false$
25:     **end if**
26:     **if** $F_P_LWorst == true$ **then**
27:         Find and store $LWorst$
28:         $F_P_LWorst = false$
29:     **end if**
30: **end for**

Algorithm 10 Improved computing performance (ICP) applied to the selection of the population update option

1: Obtain two ordered integer random numbers and one chaotic number:
2: $a = min$(rnd1,rnd2)
3: $b = max$(rnd1,rnd2)
4: $\{$Initially ch_j are randomly selected chaotic values$\}$
5: **for** $i = 5$ to 2 **do**
6: $\quad ch_j = ch_{j-1}$
7: **end for**
8: ch_1 new randomly selected chaotic value
9: Select between equations (2), (3) or (4):
10: **if** $ch_j < a$ **then**
11: $\quad x_j^{'} = ch_{1,j}x_j^{rand} + ch_{2,j}\left(x_j - ch_{3,j}x_j^{rand}\right) + ch_{4,j}\left(x_{j,best} - ch_{5,j}x_j^{rand}\right)$
12: **end if**
13: **if** $a < ch_j < b$ **then**
14: $\quad x_j^{'} = ch_{1,j}x_j^{rand} + ch_{2,j}\left(x_j - ch_{3,j}x_j^{rand}\right) + ch_{4,j}\left(x_{j,worst} - ch_{5,j}x_j^{rand}\right)$
15: **end if**
16: **if** $ch_j > b$ **then**
17: $\quad x_j^{'} = ch_{1,j}x_{j,best} + ch_{2,j}\left(x_j^{rand} - S_F x_{j,best}\right)$
18: **end if**

270 ## 4. Numerical experiments

In this section, the parallel chaotic Jaya algorithms presented in Section 3 are analysed in terms of their parallel performance and optimisation behaviour. The reference algorithm presented in [58] and the proposed parallel algorithms were implemented in the C programming language, and the GCC v.4.8.5 compiler [60] was used. The para-
275 llel approaches were designed for shared memory parallel platforms using the OpenMP API v3.1 [61]. The parallel computing platform used was equipped with two Intel Xeon X5660 processors, each of which contained six processing cores at 2.8 GHz, and hyperthreading was not activated. The performance was analysed using 18 unconstrained functions, as listed in Tables 1 and 2.

Table 1: Benchmark functions.

Id.	**Name**	**Dim. (V)**	**Domain (Min, Max)**
F1	Sphere	30	$-100, 100$
F2	SumSquares	30	$-10, 10$
F3	Beale	2	$-4.5, 4.5$
F4	Easom	2	$-100, 100$
F5	Zakharov	10	$-5, 10$
F6	Schwefel problem 1.2	10	$-100, 100$
F7	Rosenbrock	30	$-30, 30$
F8	Branin	2	$x_1 : -5, 10; x_2 : 0, 15$
F9	Bohachevsky_1	2	$-100, 100$
F10	Booth	2	$-10, 10$
F11	Michalewicz_2	2	$0, \pi$
F12	Bohachevsky_2	2	$-100, 100$
F13	Bohachevsky_3	2	$-100, 100$
F14	GoldStein-Price	2	$-2, 2$
F15	Hartman_3	3	$0, 1$
F16	Ackley	30	$-32, 32$
F17	Langermann_2	2	$0, 10$

F18	Langermann_10	10	$0, 10$

Table 2: Benchmark functions.

Id.	Function
F1	$f = \sum_{i=1}^{V} x_i^2$
F2	$f = \sum_{i=1}^{V} i x_i^2$
F3	$f = (1.5 - x_1 + x_1 x_2)^2 + (2.25 - x_1 + x_1 x_2^2)^2$ $+(2.625 - x_1 + x_1 x_2^3)^2$
F4	$f = -\cos(x_1)\cos(x_2)\exp\left(-(x_1 - \pi)^2 - (x_2 - \pi)^2\right)$
F5	$f = \sum_{i=1}^{V} x_i^2 + \left(\sum_{i=1}^{V} 0.5 i x_i\right)^2 + \left(\sum_{i=1}^{V} 0.5 i x_i\right)^4$
F6	$f = \sum_{i=1}^{V} \left(\sum_{j=1}^{i} x_j\right)^2$
F7	$f = \sum_{i=1}^{V-1} \left(100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right)$
F8	$f = \left(x_2 - \frac{5.1}{4\pi^2} x_1^2 + \frac{5}{\pi} x_1 - 6\right)^2 + 10\left(1 - \frac{1}{8\pi}\right)\cos x_1 + 10$
F9	$f = x_1^2 + 2x_2^2 - 0.3\cos(3\pi x_1) - 0.4\cos(4\pi x_2) + 0.7$
F10	$f = (x_1 - 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2$
F11	$f = -\sum_{i=1}^{2} \sin x_i \left(\sin\left(\frac{i x_i^2}{\pi}\right)\right)^{20}$
F12	$f = x_1^2 + 2x_2^2 - 0.3\cos(3\pi x_1)\cos(4\pi x_2) + 0.3$
F13	$f = x_1^2 + 2x_2^2 - 0.3\cos(3\pi x_1 + 4\pi x_2) + 0.3$
F14	$f = \left[1 + (x_1 + x_2 + 1)^2(19 - 14x_1 + 3x_1^2 - 14x_2 + 6x_1x_2 + 3x_2^2)\right]$ $\left[30 + (2x_1 - 3x_2)^2(18 - 32x_1 + 12x_1^2 + 48x_2 - 36x_1x_2 + 27x_2^2)\right]$
F15	$f = -\sum_{i=1}^{4} c_i \exp\left[-\sum_{j=1}^{3} a_{ij}(x_j - p_{ij})^2\right]$
F16	$f = -20\exp\left(-0.2\sqrt{\frac{1}{V}\sum_{i=1}^{V} x_i^2}\right) - \exp\left(\frac{1}{V}\sum_{i=1}^{V}\cos(2\pi x_i)\right) + 20 + e$

F17 F18	$f = -\sum_{i=1}^{5} c_i \left[\exp\left(-\frac{1}{\pi} \sum_{j=1}^{V} (x_j - a_{ij})^2 \right) \cos\left(\pi \sum_{j=1}^{V} (x_j - a_{ij})^2 \right) \right]$

An analysis of the computational costs of both the original Jaya algorithm and the chaotic algorithm was performed first. Table 3 shows the sequential computational times for a population size of 240, where the number of iterations was $50,000$ and the number of independent executions was 30. The results shown in Table 3 indicate that the computational cost of the chaotic algorithm is generally higher than that of the original algorithm. In addition, the sequential implementation shown in Algorithm 3 has some drawbacks in terms of efficient parallel development, such as the computing of up to seven random numbers (to calculate a and b and to select the chaotic values) and the extraction of up to five values from the chaotic map.

Although the higher computational cost of the Chaotic Jaya sequential algorithm, shown in Table 3, compared to the original sequential Jaya algorithm, is undoubtedly offset by the acceleration in convergence, the use of the chaotic map was computationally analyzed and modified to accelerate the parallel algorithms proposed in Section 3, by using the ICP technique.

Results shown in Table 4 show that the version including the improved computing performance (ICP) technique, presented in Section 3.3, significantly improved computational times of the Chaotic Jaya sequential algorithm (whose results are shown in Table 3). It can be observed in Tables 3 and 4 that the computational cost increase (with respect to the original sequential Jaya algorithm) of the Chaotic Jaya sequential algorithm is reduced by including the ICP technique. Even in some cases (negative values from table 4) the Chaotic Jaya sequential algorithm with the ICP technique is computationally less expensive than the original sequential Jaya algorithm.

The behaviour of the algorithm that includes optimisation to improve computational performance (ICP) is shown in Table 5. This table shows the number of function evaluations required to obtain an error of less than $1e-1$. The optimisation algorithm was run 10 times, and Table 5 shows the maximum, minimum and average values of the number of cost function evaluations for a population of size 240. The error for the Rosenbrock function (F7) was $1e2$, and it can be seen that the behaviour does not differ

Table 3: Comparison sequential computational times.

	Time (s.)		
	Original Jaya	Chaotic Jaya	Increment (%)
F1	186.3	1227.7	559%
F2	193.6	1241.1	541%
F3	45.7	76.1	67%
F4	43.6	79.6	83%
F5	123.5	478.8	288%
F6	353.9	1556.8	340%
F7	202.4	625.1	209%
F8	27.7	59.0	113%
F9	29.3	52.9	81%
F10	16.4	43.7	167%
F11	92.8	141.3	52%
F12	26.7	49.7	87%
F13	27.6	50.8	84%
F14	20.8	46.6	124%
F15	76.6	116.6	52%
F16	162.1	357.1	120%
F17	154.2	188.6	22%
F18	239.7	400.2	67%

Table 4: Comparison sequential computational times respect to the version with improved computing performance (ICP).

	Time (s.)		
	Original Jaya	Chaotic Jaya (ICP)	Increment (%)
F1	186.3	884.9	375%
F2	193.6	898.4	364%
F3	45.7	49.0	7%
F4	43.6	49.6	14%
F5	123.5	255.7	107%
F6	353.9	1224.3	246%
F7	202.4	240.7	19%
F8	27.7	31.0	12%
F9	29.3	25.7	-12%
F10	16.4	16.5	1%
F11	92.8	109.7	18%
F12	26.7	22.0	-17%
F13	27.6	22.7	-18%
F14	20.8	19.5	-6%
F15	76.6	77.9	2%
F16	162.1	167.2	3%
F17	154.2	169.1	10%
F18	239.7	269.5	12%

markedly from one algorithm to another. Table 5 shows slight decreases in optimisation performance in some cases, however, the high decrease in computational costs
310 compensates for these slight decreases (see Tables 3 and 4).

Table 5: Comparison of the number of cost function evaluations respect to the version with improved computing performance (ICP).

Number of functions evaluations ($error < 10e - 1$)

	Chaotic			Chaotic (ICP)		
	Average	Maximum	Minimum	Average	Maximum	Minimum
F1	5232	6240	3840	5328	6240	4560
F2	4752	5520	4080	4320	6240	3120
F3	552	960	480	552	720	480
F4	2808	4800	720	3264	9120	1920
F5	3216	4320	1680	3096	5520	960
F6	10416	12720	7440	9360	12000	7200
F7 (*)	3912	4560	2880	3936	5280	2880
F8	960	2640	480	1176	3840	480
F9	2376	3600	1680	2880	2160	1200
F10	1656	3120	720	2613	5760	480
F11	1032	2160	480	1224	2640	480
F12	2016	2640	1200	1752	2400	960
F13	1800	2640	960	1512	2160	960
F14	1848	3120	960	2256	3600	1200
F15	672	1200	480	936	2160	480
F16	4920	6240	4080	4488	6000	3360
F17	504	720	480	480	480	480
F18	480	480	480	480	480	480

Table 6 shows a significant improvement in the ratio of convergence between the chaotic algorithm and the original Jaya algorithm for the functions that required more iterations. For the functions that required fewer iterations, the improvement was significant in some cases, while in others the behaviour was similar. A more exhaustive

315 comparative analysis with respect to other algorithms can be found in [57].

Table 6: Comparison of the number of cost function evaluations respect to the original Jaya algorithm.

Number of functions evaluations ($error < 10e-1$)

	Original	Chaotic	Chaotic (ICP)
F1	532560	5232	5328
F2	441750	4752	4320
F3	780	552	552
F4	44580	2808	3264
F5	240960	3216	3096
F6	152576	10416	9360
F7 (*)	644760	3912	3936
F8	690	960	1176
F9	8880	2376	2880
F10	1710	1656	2613
F11	810	1032	1224
F12	7980	2016	1752
F13	8190	1800	1512
F14	8220	1848	2256
F15	600	672	936
F16	293550	4920	4488
F17	510	504	480
F18	480	480	480

In order to analyse the parallel behaviour of the two parallel algorithms developed here, experiments were performed with 30 independent executions and population sizes of 240, 120 and 60. Tables 7 and 8 show the speed-up achieved by CP-CJaya when no improvements were applied and when performance computing improvement was
320 included, respectively.

From both tables, it can be seen that very good acceleration is obtained for the functions of higher computational cost, even with 10 processes, for a population size

of 240. As the population size decreases, the speed-up decreases. Note, for example, that for a population of 60 individuals and 10 processes, the sub-population size is only six individuals, which makes the communication and coordination processes more expensive than the computing time.

Since the sequential reference algorithm is the same in both figures, it can be seen that the ICP use in algorithm CP-CJaya (Table 8) improves the behaviour of the parallel algorithm CP-CJaya without the ICP technique (Table 7). Note that lower intensity improvements are obtained for small sub-populations, for example, when sub-population size equals 6 (population size of 60 using 10 processes).

Tables 9 and 10 show the speed-up achieved by the parallel algorithm NCP-CJaya. Data shown in both tables have been calculated using the same sequential reference algorithm as that used in Tables 7 and 8, i.e. the Chaotic Jaya sequential algorithm. Comparing Table 9 with Table 7, and comparing Table 10 with Table 8, it can be observed that the parallel behaviour of the NCP-CJaya algorithm significantly improves the parallel behaviour of the CP-CJaya, if the ICP technique is not used (Tables 7 and 9) and if the ICP technique is used (Tables 8 and 10). Comparing Table 9 and Table 10 it can be seen that the parallel behaviour of the NCP-CJaya algorithm using the ICP technique (Table 10) improves significantly with respect to non-use of ICP (Table 9), even for a small sub-population equal to 6 (population size equal to 60 using 10 threads). As said, the speed-up results show good behaviour, even for 10 processes and small population sizes. In some cases, super-speed-up results are shown, i.e. a value greater than the number of processes. This is due both to the efficiency of the cache memory and, more importantly, to the fact that the costs of the reference algorithm and the parallel algorithm are not exactly the same.

Table 11 shows the sequential computational times of chaotic Jaya for variation in the population size, and shows that the computational cost ratio is not proportional to the size of the population. The computational cost associated with each thread will therefore depend on the size of its sub-population.

Finally, it should be noted that in most experiments with the NCP-CJaya algorithm, the speed of convergence improves as the number of processes increases. This behaviour is related to the fact that in general, the CJaya algorithm behaves better for

Table 7: Speed-up of method CP-CJaya without IPC

	Pop. Size: 240			Pop. Size: 120			Pop. Size: 60		
	N. of threads			N. of threads			N. of threads		
	2	6	10	2	6	10	2	6	10
F1	1.83	5.12	8.18	1.73	4.70	7.22	1.54	3.95	5.82
F2	1.86	5.09	8.21	1.70	4.52	6.92	1.47	3.84	5.59
F3	1.82	3.87	4.52	1.75	3.04	2.92	1.71	2.20	1.84
F4	1.76	4.24	5.19	1.95	3.63	3.47	1.59	2.30	1.99
F5	1.70	4.54	7.26	1.24	3.31	4.69	0.84	2.04	2.64
F6	1.92	5.37	8.61	1.92	5.35	8.29	1.81	4.80	7.21
F7	1.97	5.43	8.27	1.95	5.19	7.30	1.87	4.81	6.13
F8	1.88	4.08	4.62	1.75	2.94	2.68	1.53	1.99	1.59
F9	1.85	3.85	4.35	1.74	2.82	2.49	1.79	1.85	1.40
F10	1.82	3.65	3.81	1.70	2.54	2.09	1.43	1.54	1.20
F11	1.77	3.89	5.00	1.73	3.39	3.79	1.66	2.73	2.66
F12	1.81	3.80	4.16	1.72	2.69	2.33	1.48	1.79	1.37
F13	1.86	3.85	4.24	1.73	2.80	2.38	1.54	1.75	1.36
F14	1.85	3.78	4.02	1.71	2.60	2.19	1.54	1.79	1.33
F15	1.88	4.59	6.08	1.84	3.75	4.13	1.70	3.13	2.94
F16	1.96	5.28	7.87	1.92	4.90	6.66	1.87	4.48	5.44
F17	1.95	5.07	7.13	1.91	4.35	5.10	1.80	3.54	3.62
F18	1.94	5.26	7.94	1.94	4.90	6.73	1.89	4.35	5.30

Table 8: Speed-up of method CP-CJaya with IPC

	Pop. Size: 240			Pop. Size: 120			Pop. Size: 60		
	N. of threads			N. of threads			N. of threads		
	2	6	10	2	6	10	2	6	10
F1	2.52	6.62	10.59	2.33	6.10	9.13	2.03	5.07	7.11
F2	2.55	6.76	10.55	2.25	5.89	8.69	1.96	4.83	6.74
F3	2.71	5.19	5.16	2.70	3.73	3.31	2.20	2.48	1.85
F4	2.90	5.68	6.49	2.95	4.70	4.07	2.32	2.77	2.15
F5	2.23	6.06	9.14	1.70	4.19	5.83	1.09	2.44	3.02
F6	2.44	6.84	10.81	2.52	6.66	10.35	2.30	5.96	8.42
F7	4.92	12.13	17.84	4.88	10.91	12.70	4.53	8.89	9.65
F8	3.39	5.99	5.92	3.36	3.77	3.02	2.41	2.31	1.69
F9	3.51	5.96	5.33	3.09	3.51	2.81	2.42	2.14	1.48
F10	3.95	5.70	4.92	3.37	3.30	2.39	2.44	1.88	1.30
F11	2.28	4.82	5.88	2.37	4.12	4.32	2.08	3.17	2.88
F12	3.67	5.84	5.42	3.15	3.40	2.67	2.50	2.02	1.46
F13	3.55	6.01	5.54	3.26	3.67	2.74	2.40	1.95	1.43
F14	3.94	6.01	5.32	3.27	3.46	2.52	2.64	2.07	1.46
F15	2.88	6.53	7.83	2.85	4.95	4.77	2.49	3.78	3.23
F16	4.08	10.09	13.98	3.97	8.66	10.09	3.66	6.92	7.36
F17	2.20	5.82	7.96	2.26	4.81	5.69	2.11	3.84	3.79
F18	2.97	7.75	11.30	2.94	6.96	8.84	2.81	5.84	6.29

Table 9: Speed-up of method NCP-CJaya without IPC

	Pop. Size: 240			Pop. Size: 120			Pop. Size: 60		
	N. of threads			N. of threads			N. of threads		
	2	6	10	2	6	10	2	6	10
F1	2.05	6.64	12.66	2.08	8.70	17.65	2.11	8.65	14.29
F2	2.00	6.62	12.71	2.14	8.71	16.85	2.29	8.71	15.61
F3	1.99	5.19	8.63	1.74	5.17	8.40	1.75	5.01	8.30
F4	1.91	5.27	9.28	1.90	5.70	9.28	1.96	5.31	8.68
F5	2.11	7.37	18.07	2.17	8.72	14.77	2.10	7.33	11.97
F6	1.99	5.78	9.92	1.99	7.47	16.76	1.98	10.59	17.60
F7	2.00	5.47	9.29	1.98	5.30	9.55	1.82	5.68	9.39
F8	1.97	5.14	8.55	1.75	5.12	8.49	1.77	5.16	8.43
F9	1.91	5.09	8.37	1.89	5.15	8.53	1.85	5.44	8.93
F10	1.75	5.21	9.09	1.78	5.44	8.80	1.79	5.28	8.53
F11	1.93	5.21	8.59	1.75	4.80	8.53	1.74	5.10	8.48
F12	1.75	4.94	8.07	1.84	5.18	8.55	1.75	5.23	8.05
F13	1.78	5.20	8.53	1.77	5.22	8.48	1.74	5.16	8.59
F14	1.76	5.20	8.82	1.77	5.23	8.56	1.83	5.21	8.37
F15	1.92	5.18	8.67	1.76	5.21	8.63	1.78	5.20	8.92
F16	1.97	5.23	8.71	1.82	5.25	9.03	1.75	5.17	8.63
F17	1.90	5.22	8.69	2.00	5.23	8.94	1.75	5.22	9.13
F18	1.99	5.31	8.71	1.93	5.20	8.65	1.77	5.27	8.76

Table 10: Speed-up of method NCP-CJaya with IPC

	Pop. Size: 240			Pop. Size: 120			Pop. Size: 60		
	N. of threads			N. of threads			N. of threads		
	2	6	10	2	6	10	2	6	10
F1	2.96	10.23	18.12	3.22	16.35	42.36	4.52	22.80	32.15
F2	3.01	11.00	20.40	3.30	16.87	43.03	4.39	20.75	31.70
F3	3.08	8.90	9.37	2.73	7.83	12.34	2.70	7.40	11.25
F4	3.61	10.39	9.80	3.48	9.40	14.29	2.81	7.92	12.41
F5	4.01	20.57	21.96	6.17	19.47	32.57	4.39	12.50	18.45
F6	2.53	7.46	11.42	2.70	10.56	36.06	3.25	17.62	27.63
F7	5.23	14.53	15.47	4.94	13.93	20.35	4.91	11.98	17.58
F8	3.84	10.38	10.05	3.30	9.21	14.63	3.29	8.80	13.40
F9	4.04	10.91	11.02	3.54	9.60	15.52	3.30	9.44	14.60
F10	5.11	14.10	11.99	4.59	11.54	17.65	4.07	10.04	14.52
F11	2.63	7.49	8.98	2.64	6.74	10.94	2.49	6.46	10.27
F12	4.33	11.37	13.67	3.77	10.17	16.58	3.64	9.45	14.16
F13	4.11	11.30	11.14	3.72	10.17	16.54	3.61	9.81	14.60
F14	4.62	13.25	11.63	4.44	11.26	17.31	3.98	10.38	15.25
F15	3.07	8.54	8.24	3.08	8.27	13.43	2.63	7.35	11.81
F16	4.34	12.05	11.79	3.79	10.33	16.45	4.05	9.95	15.31
F17	2.38	6.68	7.15	2.33	6.19	9.97	2.06	5.98	9.85
F18	3.03	8.65	9.88	2.97	7.88	12.76	2.89	7.56	12.12

Table 11: Sequential computational times (s.).

	Population size		
	240	120	60
Cjaya	895.82	394.71	156.81
Increment (respect pop. size = 60)	5.71	2.52	
Increment (respect pop. size = 120)	2.27		
Cjaya (ICP)	1227.48	573.55	234.50
Increment (respect pop. size = 60)	5.23	2.45	
Increment (respect pop. size = 120)	2.14		

small populations. In fact, it is able to obtain an optimal value of the function with populations of only six individuals. Table 12 demonstrates this behaviour, and shows the maximum number of iterations needed to obtain an optimal value with a tolerance of $1e-1$.

5. Conclusions

Two parallel proposals are presented here for accelerating the heuristic optimisation algorithm Jaya using a chaotic 2D map. For both proposals, a strategy to reduce the computational cost by varying the strategy of use of the chaotic map is analysed, and the results are validated with respect to the optimisation behaviour. These parallel proposals are analysed in detail, both at the level of parallel performance and at the level of optimisation behaviour. In addition to verifying that the chaotic map used improves the ratio of convergence, we demonstrate that parallel algorithms can even improve the convergence speed. Although the CP-CJaya algorithm offers less scalability, the NCP-CJaya algorithm offers optimal scalability. In future work, we will develop hybrid versions at the level of chaotic maps and the number and size of the sub-populations in order to accelerate convergence without losing parallel efficiency.

Table 12: Maximum number of cost function evaluations with improved computing performance (ICP). Tolerance $1e-1$

	Maximum number of functions evaluations		
	Pop. Size = 240	Pop. Size = 120	Pop. Size = 60
F1	6240	3360	1020
F2	6240	2040	900
F3	720	240	120
F4	9120	1800	480
F5	5520	1680	1140
F6	12000	3720	1560
F7 (*)	5280	2520	1200
F8	3840	600	420
F9	2160	720	540
F10	5760	4080	900
F11	2640	360	1140
F12	2400	1200	540
F13	2160	720	300
F14	3600	2760	10680
F15	2160	1320	660
F16	6000	2280	960
F17	480	480	120
F18	480	240	120

370 ## Acknowledgments

This research was supported by the Spanish Ministry of Science, Innovation and Universities under Grant RTI2018-098156-B-C54 co-financed by FEDER funds, and by the Spanish Ministry of Economy and Competitiveness under Grant TIN2017-89266-R, co-financed by FEDER funds (MINECO/FEDER/UE).

375 ## References

[1] M.-H. Lin, J.-F. Tsai, C.-S. Yu, A review of deterministic optimization methods in engineering and management, Mathematical Problems in Engineering 2012 (Article ID 756023) (2012) 15. doi:10.1155/2012/756023.

[2] R. Poli, J. Kennedy, T. Blackwell, Particle swarm optimization, Swarm Intelli-
380 gence 1 (1) (2007) 33–57. doi:10.1007/s11721-007-0002-0.

[3] D. Karaboga, B. Basturk, On the performance of artificial bee colony (abc) algorithm, Appl. Soft Comput. 8 (1) (2008) 687–697. doi:10.1016/j.asoc.2007.05.007.

[4] M. Eusuff, K. Lansey, F. Pasha, Shuffled frog-leaping algorithm: a memetic meta-
385 heuristic for discrete optimization, Engineering Optimization 38 (2) (2006) 129–154. doi:10.1080/03052150500384759.

[5] M. Dorigo, G. Di Caro, New ideas in optimization, McGraw-Hill Ltd., UK, Maidenhead, UK, England, 1999, Ch. The Ant Colony Optimization Meta-heuristic, pp. 11–32.
390 URL http://dl.acm.org/citation.cfm?id=329055.329062

[6] H.-P. Schwefel, Evolutionsstrategie und numerische Optimierung, Dr.-Ing. Thesis, Technical University of Berlin, Department of Process Engineering (1975).

[7] J. R. Koza, Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems, Tech. rep., Stanford, CA, USA
395 (1990).

[8] T. Bck, G. Rudolph, H. paul Schwefel, Evolutionary programming and evolution strategies: Similarities and differences, in: In Proceedings of the Second Annual Conference on Evolutionary Programming, 1997, pp. 11–22.

[9] Y. Xin-She, Firefly algorithm, lvy flights and global optimization, Research and Development in Intelligent Systems XXVI (2009) 209–218doi:10.1007/978-1-84882-983-1_15.

[10] E. Rashedi, H. Nezamabadi-pour, S. Saryazdi, Gsa: A gravitational search algorithm, Information Sciences 179 (13) (2009) 2232 – 2248, special Section on High Order Fuzzy Sets. doi:10.1016/j.ins.2009.03.004.

[11] H. Ma, D. Simon, P. Siarry, Z. Yang, M. Fei, Biogeography-based optimization: A 10-year review, IEEE Transactions on Emerging Topics in Computational Intelligence 1 (5) (2017) 391–407. doi:10.1109/TETCI.2017.2739124.

[12] A. Ahrari, A. A. Atai, Grenade explosion methoda novel tool for optimization of multimodal functions, Applied Soft Computing 10 (4) (2010) 1132 – 1140. doi:10.1016/j.asoc.2009.11.032.

[13] J. H. Holland, Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence, MIT Press, 1992.

[14] J. D. Farmer, N. H. Packard, A. S. Perelson, The immune system, adaptation, and machine learning, Phys. D 2 (1-3) (1986) 187–204. doi:10.1016/0167-2789(81)90072-5.

[15] K. V. Price, New ideas in optimization, McGraw-Hill Ltd., UK, Maidenhead, UK, England, 1999, Ch. An Introduction to Differential Evolution, pp. 79–108.
URL http://dl.acm.org/citation.cfm?id=329055.329069

[16] L. Ingber, Simulated annealing: Practice versus theory, Mathematical and Computer Modelling 18 (11) (1993) 29 – 57. doi:10.1016/0895-7177(93)90204-C.

[17] F. Glover, Interfaces in computer science and operations research, Springer, Boston, MA, 1997, Ch. Tabu Search and Adaptive Memory Programming Advances, Applications and Challenge, pp. 1–75.
URL http://dl.acm.org/citation.cfm?id=329055.329069

[18] R. V. Rao, V. Savsani, D. Vakharia, Teaching-learning-based optimization: A novel method for constrained mechanical design optimization problems, Computer-Aided Design 43 (3) (2011) 303–315. doi:10.1016/j.cad.2010.12.015.

[19] J. H. Kim, Harmony search algorithm: A unique music-inspired algorithm, Procedia Engineering 154 (2016) 1401 – 1405, 12th International Conference on Hydroinformatics (HIC 2016) - Smart Water for the Future. doi:10.1016/j.proeng.2016.07.510.

[20] S. Mishra, P. K. Ray, Power quality improvement using photovoltaic fed dstatcom based on jaya optimization, IEEE Transactions on Sustainable Energy 7 (4) (2016) 1672–1680. doi:10.1109/TSTE.2016.2570256.

[21] C. Huang, L. Wang, R. S. Yeung, Z. Zhang, H. S. Chung, A. Bensoussan, A prediction model-guided Jaya algorithm for the PV system maximum power point tracking, IEEE Transactions on Sustainable Energy 9 (1) (2018) 45–55. doi:10.1109/TSTE.2017.2714705.

[22] B. Akay, D. Karaboga, Artificial bee colony algorithm for large-scale problems and engineering design optimization, Journal of Intelligent Manufacturing 3 (4) (2010) 1001–1014. doi:10.1007/s10845-010-0393-4.

[23] K. Abhishek, V. R. Kumar, S. Datta, S. S. Mahapatra, Application of jaya algorithm for the optimization of machining performance characteristics during the turning of cfrp (epoxy) composites: comparison with tlbo, ga, and ica, Engineering with Computers (2016) 1–19doi:10.1007/s00366-016-0484-8.

[24] A. Choudhary, M. Kumar, D. R. Unune, Investigating effects of resistance wire heating on aisi 1023 weldment characteristics during asaw, Materials and Man-

ufacturing Processes 33 (7) (2018) 759–769. doi:10.1080/10426914.2017.1415441.

[25] D. Dinh-Cong, H. Dang-Trung, T. Nguyen-Thoi, An efficient approach for optimal sensor placement and damage identification in laminated composite structures, Advances in Engineering Software 119 (2018) 48 – 59. doi:10.1016/j.advengsoft.2018.02.005.

[26] S. P. Singh, T. Prakash, V. Singh, M. G. Babu, Analytic hierarchy process based automatic generation control of multi-area interconnected power system using Jaya algorithm, Engineering Applications of Artificial Intelligence 60 (2017) 35–44. doi:10.1016/j.engappai.2017.01.008.

[27] H. Li, K. Ge Li, J. An, K. Ge Li, An online and scalable model for generalized sparse non-negative matrix factorization in industrial applications on multi-GPU, IEEE Transactions on Industrial Informatics (2019) 1–1doi:10.1109/TII.2019.2896634.

[28] H. Li, K. Li, J. An, K. Li, MSGD: A novel matrix factorization approach for large-scale collaborative filtering recommender systems on GPUs, IEEE Transactions on Parallel and Distributed Systems 29 (7) (2018) 1530–1544. doi:10.1109/TPDS.2017.2718515.

[29] H. Li, K. Li, J. An, W. Zheng, K. Li, An efficient manifold regularized sparse non-negative matrix factorization model for large-scale recommender systems on GPUs, Information Sciences 496 (2019) 464 – 484. doi:10.1016/j.ins.2018.07.060.

[30] N. Medina-Rodriguez, O. Montiel-Ross, R. Sepulveda, O. Castillo, Tool path optimization for computer numerical control machines based on parallel aco, Engineering Letters 20 (2012) 101–108.

[31] C. C. Columbus, S. P. Simon, A parallel abc for security constrained economic dispatch using shared memory model, in: 2012 International Conference on

Power, Signals, Controls and Computation, 2012, pp. 1–6. doi:10.1109/EPSCICON.2012.6175239.

480 [32] N. C. Cruz, J. L. Redondo, J. D. Álvarez, M. Berenguel, P. M. Ortigosa, A parallel teaching–learning-based optimization procedure for automatic heliostat aiming, The Journal of Supercomputing 73 (1) (2017) 591–606. doi:10.1007/s11227-016-1914-5.
URL https://doi.org/10.1007/s11227-016-1914-5

485 [33] M. Z. Ali, N. H. Awad, P. N. Suganthan, R. M. Duwairi, R. G. Reynolds, A novel hybrid cultural algorithms framework with trajectory-based search for global numerical optimization, Information Sciences 334-335 (2016) 219 – 249. doi:10.1016/j.ins.2015.11.032.

[34] N. H. Awad, M. Z. Ali, P. N. Suganthan, R. G. Reynolds, Cade: A hybridization of
490 cultural algorithm and differential evolution for numerical optimization, Information Sciences 378 (2017) 215 – 241. doi:10.1016/j.ins.2016.10.039.

[35] Y. Bai, S. Xiao, C. Liu, B. Wang, A hybrid iwo/pso algorithm for pattern synthesis of conformal phased arrays, IEEE Transactions on Antennas and Propagation 61 (4) (2013) 2328–2332. doi:10.1109/TAP.2012.2231936.

495 [36] M. Ghasemi, S. Ghavidel, S. Rahmani, A. Roosta, H. Falah, A novel hybrid algorithm of imperialist competitive algorithm and teaching learning algorithm for optimal power flow problem with non-smooth cost functions, Eng. Appl. Artif. Intell. 29 (2014) 54–69. doi:10.1016/j.engappai.2013.11.003.

[37] N. Zhou, A. Zhang, F. Zheng, L. Gong, Novel image compressionencryption
500 hybrid algorithm based on key-controlled measurement matrix in compressive sensing, Optics & Laser Technology 62 (2014) 152 – 160. doi:10.1016/j.optlastec.2014.02.015.

[38] M. Majumdar, T. Mitra, K. Nishimura, Optimization and Chaos, Springer, New York, 2000.

505 [39] E. Ott, Frontmatter, 2nd Edition, Cambridge University Press, 2002, pp. i–iv.

[40] A. Rezaee Jordehi, A chaotic-based big bangbig crunch algorithm for solving global optimisation problems, Neural Computing and Applications 25 (2014) 1329–1335. doi:10.1007/s00521-014-1613-1.

[41] A. Gandomi, X.-S. Yang, S. Talatahari, A. Alavi, Firefly algorithm with chaos, 510 Communications in Nonlinear Science and Numerical Simulation 18 (1) (2013) 89 – 98. doi:10.1016/j.cnsns.2012.06.009.

[42] S. Gokhale, V. Kale, An application of a tent map initiated chaotic firefly algorithm for optimal overcurrent relay coordination, International Journal of Electrical Power & Energy Systems 78 (2016) 336 – 342. doi:10.1016/j. 515 ijepes.2015.11.087.

[43] Z. S. Ma, Chaotic populations in genetic algorithms, Applied Soft Computing 12 (8) (2012) 2409 – 2424. doi:10.1016/j.asoc.2012.03.001.

[44] X. F. Yan, D. Z. Chen, S. X. Hu, Chaos-genetic algorithms for optimizing the operating conditions based on RBF-PLS model, Computers & Chemical En- 520 gineering 27 (10) (2003) 1393 – 1404. doi:10.1016/S0098-1354(03)00074-7.

[45] W.-C. Hong, Traffic flow forecasting by seasonal SVR with chaotic simulated annealing algorithm, Neurocomputing 74 (12) (2011) 2096 – 2107. doi:10.1016/j.neucom.2010.12.032.

525 [46] J. Mingjun, T. Huanwen, Application of chaos in simulated annealing, Chaos, Solitons & Fractals 21 (4) (2004) 933 – 941. doi:10.1016/j.chaos.2003.12.032.

[47] J. Saremi, S. Mirjalili, A. Lewisn, Biogeography-based optimisation with chaos, Neural Computing and Applications 25 (5) (2014) 1077 – 1097. doi:10.530 1007/s00521-014-1597-x.

[48] X. Wang, H. Duan, A hybrid biogeography-based optimization algorithm for job shop scheduling problem, Computers & Industrial Engineering 73 (2014) 96 – 114. doi:10.1016/j.cie.2014.04.006.

[49] D. Jia, G. Zheng, M. K. Khan, An effective memetic differential evolution algorithm based on chaotic local search, Information Sciences 181 (15) (2011) 3175 – 3187. doi:10.1016/j.ins.2011.03.018.

[50] C. Peng, H. Sun, J. Guo, G. Liu, Dynamic economic dispatch for wind-thermal power system using a novel bi-population chaotic differential evolution algorithm, International Journal of Electrical Power & Energy Systems 42 (1) (2012) 119 – 126. doi:10.1016/j.ijepes.2012.03.012.

[51] B. Alatas, Chaotic bee colony algorithms for global numerical optimization, Expert Systems with Applications 37 (8) (2010) 5682 – 5687. doi:10.1016/j.eswa.2010.02.042.

[52] S. Gao, C. Vairappan, Y. Wang, Q. Cao, Z. Tang, Gravitational search algorithm combined with chaos for unconstrained numerical optimization, Applied Mathematics and Computation 231 (2014) 48 – 62. doi:10.1016/j.amc.2013.12.175.

[53] R. V. Rao, A. Saroj, A self-adaptive multi-population based Jaya algorithm for engineering optimization, Swarm and Evolutionary Computation 37 (2017) 1 – 26. doi:10.1016/j.swevo.2017.04.008.

[54] H. Migallón, A. Jimeno-Morenilla, J.-L. Sánchez-Romero, H. Rico, R. V. Rao, Multipopulation-based multi-level parallel enhanced Jaya algorithms, The Journal of Supercomputing 75 (2019) 1697 – 1716. doi:10.1007/s11227-019-02759-z.

[55] P. D. Michailidis, An efficient multi-core implementation of the Jaya optimisation algorithm, International Journal of Parallel, Emergent and Distributed Systems 0 (0) (2017) 1–33. doi:10.1080/17445760.2017.1416387.

[56] A. García-Monzó, H. Migallón, A. Jimeno-Morenilla, J.-L. Sánchez-Romero, H. Rico, R. V. Rao, Efficient subpopulation based parallel TLBO optimization algorithms, Electronics 8 (1). doi:10.3390/electronics8010019.

[57] A. Farah, A. Belazi, A novel chaotic Jaya algorithm for unconstrained numerical optimization, Nonlinear Dynamics 93 (2018) 1451 – 1480. doi:10.1007/s11071-018-4271-5.

[58] R. V. Rao, Jaya: A simple and new optimization algorithm for solving constrained and unconstrained optimization problems, International Journal of Industrial Engineering Computations 7 (2016) 19–34. doi:10.5267/j.ijiec.2015.8.004.

[59] R. V. Rao, G. Waghmare, A new optimization algorithm for solving complex constrained design optimization problems, Engineering Optimization 49 (1) (2017) 60–83. doi:10.1080/0305215X.2016.1164855.

[60] Free Software Foundation, Inc., GCC, the gnu compiler collection, https://www.gnu.org/software/gcc/index.html.

[61] OpenMP Architecture Review Board, OpenMP Application Program Interface, version 3.1, http://www.openmp.org.